



Criando os Ambientes de Banco de Dados, Backend e Frontend no Docker

Chegou a hora de unificarmos tudo o que vimos até aqui, na disciplina de Prática Integradora Tecnologias Disruptivas. Para isso, desenvolveremos, enfim, o nosso projeto integrado, fullstack, composto por um banco de dados (onde utilizaremos o postgresql), pelo backend (API Restful escrita com Spring Boot) e pelo frontend (aplicação Web escrita com React.js). Entretanto, antes de vermos os códigos do back e front end em si, veremos, neste módulo, como preparar os nossos ambientes. Para isso, utilizaremos Docker – a exemplo do que já foi feito na disciplina Tecnologias Inovadoras de Desenvolvimento de Sistemas. Portanto, pode ser uma boa ideia você revisar essa disciplina, pois utilizaremos aqui os mesmos conceitos já vistos e aplicados na mesma.

Antes de continuarmos, vamos à configuração do ambiente e também aos pré-requisitos necessários para a continuidade do processo como um todo. Em meu ambiente, estou usando o Docker em uma distribuição Linux Ubuntu. Entretanto, você poderá trabalhar em ambiente Windows ou outro de sua preferência. Inclusive, essa é uma das grandes vantagens do Docker. Além disso, listo abaixo algumas versões, de aplicativos que precisaremos, que uso aqui:

- Java: openjdk 11.0.16



- Apache Maven: 3.6.3

- Docker: 20.10.17



Desses aplicativos acima, os mais importantes são o Docker e também o Maven. O primeiro, tendo em vista a compatibilidade com as versões de dockerfile ou outros recursos específicos de versão. O segundo, caso queiramos compilar nosso backend fora do processo de “dockerização” (na disciplina de Tecnologias Inovadoras, o processo de build foi realizado durante a dockerização). Isso porque a versão de Java, e até mesmo a do Maven, podem – e são – definidas diretamente no dockerfile.

Outro ponto importante, antes de continuarmos, é definirmos uma estrutura de pastas para nossas aplicações e arquivos referentes ao Docker. Veja abaixo como ficou a minha estrutura:

```
/aplicacao
    /backend/
    /bancodedados/
        /create_tables.sql
        /data/
    /dockerfiles/
        /Dockerfile.SpringBoot
        /Dockerfile.React
    /frontend/
    docker-compose.yml
```

.dockerignore



A respeito da estrutura acima, seguem alguns comentários adicionais:

- A pasta raíz é a pasta "aplicacao". Tal pasta contém subpastas e também os arquivos docker-compose.yml e .dockerignore, onde definiremos os nossos serviços e demais configurações dos contêineres;

- O arquivo Docker-compose.yml contém a configuração dos serviços Docker que subiremos;

- O arquivo .dockerignore contém arquivos que deverão ser ignorados no processo de dockerização, inclusive no eventual mapeamento de volumes, etc. O destaque aqui é a inclusão da pasta node_modules, do frontend, para que não seja considerada durante o processo de cópia do código-fonte;

- A pasta bancodedados contém o sql para construção da estrutura do banco de dados – tabelas, colunas e relacionamentos e será executado durante a subida do Docker. Já a pasta data é um volume, mapeado para a pasta postgres que armazena os dados. O uso dessa pasta permite que os dados não sejam perdidos durante eventuais recriações das imagens. Por outro lado, você poderá ter problemas de permissionamento na recriação de imagens. Caso isso aconteça, serão necessários alguns passos extras, como a alteração de propriedade do arquivo ou até mesmo a sua remoção para posterior recriação;

- Na pasta dockerfiles ficarão os arquivos de configuração específicos do back e do frontend;

- As pastas backend e frontend armazenarão, como os nomes indicam, os códigos-fonte de tais aplicações.

Uma última coisa, antes de continuarmos: como mencionado acima, poderemos executar a configuração do backend e subida da API de duas formas:

1. Realizando o build do código fora do Docker, gerando assim a pasta target e, dentro dela, além de outros arquivos, o .jar de nossa aplicação;

2. Realizando o build do código dentro do processo de dockerização. Nesse caso, precisaremos copiar todo o nosso código para dentro do container Docker e, então, realizar o build. No passo 1, só precisaremos copiar o conteúdo da pasta target.

Em relação a essas duas opções, não há uma certa e outra errada. São duas formas válidas de realizar o mesmo processo. Considerando que a forma 2 foi vista na outra disciplina, utilizarei aqui a forma 1. Entretanto, você pode ver abaixo os dois arquivos Dockerfile utilizados em ambos os casos – reforçando que o docker-compose.yml é o mesmo.



```
#
# Build Stage
#
FROM maven:3.6.3-openjdk-11-slim AS build

COPY backend/target /home/app/target

#Apagando os arquivos, exceto o .jar
RUN bash -O extglob -c 'rm -rfv /home/app/target/!(*.jar)'



#
# Package stage
#
FROM openjdk:11-slim
COPY --from=build /home/app/target/*.jar /usr/local/lib/springbootapi.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/local/lib/springbootapi.jar"]
```

Dockerfile.Spring.Boot – Opção 1

Utilizando essa primeira opção, você precisará compilar o código antes de subir o container. Para isso, e estando na pasta raíz do backend (onde fica o arquivo "pom.xml"), execute o comando:

./mvnw clean package -DskipTests

Esse comando acima é para o ambiente Windows. A flag -DskipTests é usada para não executar os testes durante o build, que falhariam por conta do banco de dados.

Estando no ambiente Linux, o comando de build deverá ser esse:

mvn clean package -DskipTests



```
#
# Build Stage
#
FROM maven:3.6.3-openjdk-11-slim AS build

COPY backend/src /home/app/src
COPY backend/pom.xml /home/app
RUN mvn -f /home/app/pom.xml clean package -DskipTests

#
# Package stage
#
FROM openjdk:11-slim
COPY --from=build /home/app/target/*.jar /usr/local/lib/springbootapi.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/local/lib/springbootapi.jar"]
```

Dockerfile.Spring.Boot – Opção 2

Perceba que na opção 2, no estágio de Build, terceiro e último passo, após cópia da pasta src do host para dentro do container, o projeto é “buildado”, usando o mesmo comando (do maven), mostrado acima.

Em relação ao front, o Dockerfile é bastante simples e também parecido com o utilizado na Disciplina de Tecnologias Inovadoras. Seu conteúdo pode ser visto a seguir:



```
FROM node:slim

#Define o diretorio de trabalho dentro do container
WORKDIR /app

#Opcional: instala o bash e o procps
RUN /bin/sh -c "apt-get install bash"
RUN apt-get update && apt-get install -y procps

#Copia o arquivo de dependencias para o workdir
COPY frontend/package.json ./

#Instala as dependencias definidas no package.json
RUN npm install

#Copia o codigo da aplicacao do host para o container
COPY frontend/ .

EXPOSE 3000

CMD ["npm", "start"]
```

Dockerfile.React

No arquivo acima cada passo é comentado, facilitando o entendimento do que está acontecendo a cada linha. De forma geral, é definido um diretório de trabalho, dentro do container. A seguir, o arquivo de controle de dependências do projeto, o package.json, é copiado para dentro desse diretório. A seguir, as dependências são instaladas. Aqui, poderíamos até mesmo copiar a pasta node_modules do nosso projeto para dentro do container. Entretanto, esse processo seria mais demorado do que copiar apenas o package e depois o restante do código, da forma como estamos fazendo. Inclusive, incluímos a pasta node_modules no .dockerignore, na raíz do projeto, para que não seja copiada na instrução “COPY frontend”. A última etapa no dockerfile do front é executar a aplicação rodando o comando “npm start”.

Seguindo o fluxo acima, nesse ponto você já tem uma visão geral do processo de dockerização de nossos ambientes. Temos, inclusive, os arquivos de configuração necessários para subirmos todos esses ambientes através do Docker-compose. Logo, já podemos testar o que fizemos até aqui. Mas, antes disso, é importante frisar: o serviço de banco de dados subirá sem problemas, criando tabelas, colunas e relacionamentos, inclusive. Contudo, para subirmos o front e o back, precisamos das aplicações em si. Do contrário será gerado erros em ambos os serviços. Como ainda não codificamos a nossa aplicação completa, você pode fazer o seguinte para testar a criação dos ambientes via Docker:

1. Criar uma nova aplicação React na pasta frontend, conforme estrutura apresentada acima. Lembrando que essa pasta deverá ser a pasta raiz do front;

2. Criar uma nova API Spring Boot na pasta backend – que também deverá ser a pasta raiz de tal aplicação. Além disso, você precisará mapear, ao menos, uma Entidade no package de entidades – pasta entity. Sem isso o Spring gerará um erro quando estiver inicializando.

O primeiro passo poderá ser feito com o seguinte comando, a partir da pasta raiz de todo o projeto (em nosso caso, a pasta “aplicacao”:

npx create-react-app frontend

Esse comando gerará uma nova aplicação React contendo um componente default no arquivo src/App.js . Para fins de teste, esse componente será o suficiente para testarmos o front.

Para criar o back, utilize o site start.spring.io. Você pode se basear na configuração constante na imagem abaixo:



spring initializr

Project
Maven Project   Gradle Project

Language
Java   Kotlin   Groovy

Spring Boot
3.0.0 (SNAPSHOT)   3.0.0 (M5)   2.7.5 (SNAPSHOT)   2.7.4
2.6.13 (SNAPSHOT)   2.6.12

Project Metadata

Group: com.projeto
Artifact: integrado
Name: integrado
Description: Projeto FullStack
Package name: com.projeto.integrado
Packaging: Jar   War
Java: 19   17   11   8

Dependencies — ADD DEPENDENCIES... CTRL + B

Spring Boot DevTools — DEVELOPER TOOLS
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Spring Web — WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security — SECURITY
Highly customizable authentication and access-control framework for Spring applications.

Spring Data JPA — SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

PostgreSQL Driver — SQL
A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

Figura 1: Template de Projeto Spring Boot

Após realizar o download do template, copie seu conteúdo para a pasta "backend", dentro da pasta raiz "aplicacao".

Agora, crie a estrutura de pastas/pacotes da API, deixando-a dessa forma:

- config
- controller

- entity



- repositor
- security
- service

Aqui, uma vez que defini lá no start.spring.io o Group como "com.projeto" e o Artifact como "integrado", o package ficou como "com.projeto.integrado" e, consequentemente, a pasta source completa ficou assim:

backend\src\main\java\com\projeto\integrado

E, abaixo da última pasta na hierarquia acima, as pastas entity, repository, etc.

Veja como ficou a minha pasta backend ao final de todo o processo:

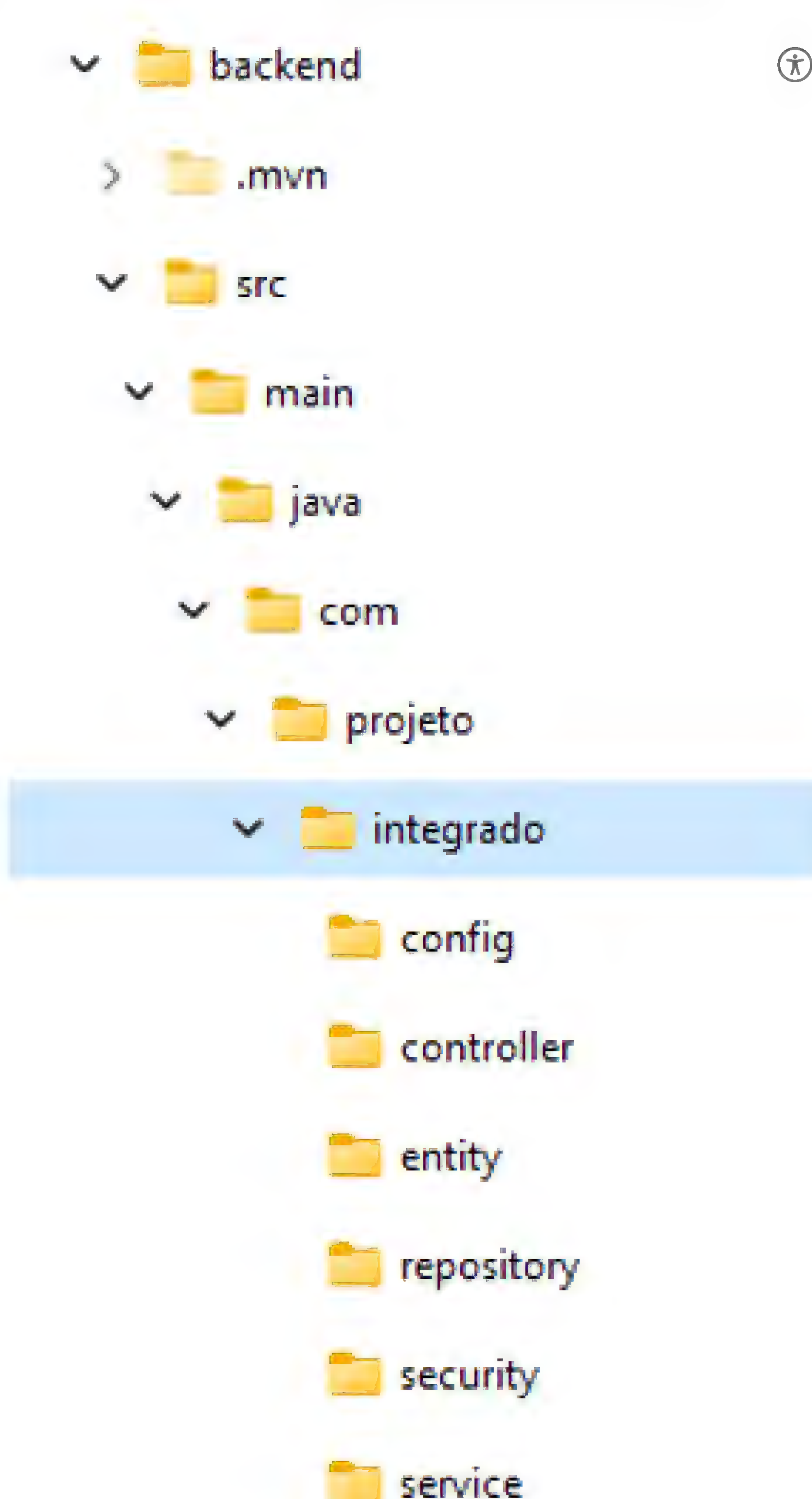

backend
.mvn
src
main
java
com
projeto
integrado
config
controller
entity
repository
security
service



Figura 2: Estrutura de pastas do projeto de backend

Agora, com a estrutura de pastas e o esqueleto da API prontos, podemos criar uma Entidade a fim de testar a aplicação. Para isso, vamos mapear a tabela "projeto", criando a entidade Projeto, na pasta "entity". Seu código deverá ficar assim:

```
package com.projeto.integrado.entity;

//imports ...

@Entity
@Table(name = "projeto")
public class Projeto {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "projeto_id")
        private Integer projetoId;

        @Column(name = "projeto_nome")
        private String projetoNome;

        @Column(name = "projeto_descricao")
        private String projetoDescricao;

        @Column(name = "projeto_inicio")
        private Instant projetoInicio;

        @Column(name = "projeto_fim")
        private Instant projetoFim;

        @Column(name = "projeto_status")
        private Boolean projetoStatus;

        //get´s e set´s
}
```

Entidade Projeto.java



Para não gerar muitas linhas, repare que toda a parte de import e também os métodos get e set foram omitidos no código acima. Além disso, embora funcional, além de servir ao nosso propósito momentâneo, que é testar a API spring dentro do container Docker, a entidade acima ainda não é a versão final, pois não contém os mapeamentos das relações, por exemplo. Veremos isso em outro módulo. Por ora, com a entidade acima criada, podemos continuar.

Chegou a hora de subir todo o nosso ambiente Docker, composto pelo banco de dados, api de backend e aplicação de front end. O docker-compose.yml contendo cada um desses serviços ficou assim:



```yaml
version: '3.1'

services:
  backend:
    image: 'docker-spring-boot:latest'
    container_name: backend
    build:
      context: ../descomplica
      dockerfile: ./dockerfiles/Dockerfile.Spring.Boot
    tty: true
    depends_on:
      db_postgresql:
        condition: service_healthy
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db_postgresql:5432/bd_projeto_integrado
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=password-bd-projeto-integrado
      - SPRING_JPA_HIBERNATE_DDL_AUTO=update
    restart: unless-stopped
    ports:
        - 18080:8080
    networks:
        - projeto-integrado-network


  db_postgresql:
    image: 'postgres:13.1-alpine'
    container_name: db_postgresql
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password-bd-projeto-integrado
      - POSTGRES_DB=bd_projeto_integrado
    restart: always
    volumes:
        - ./bancodedados/data:/var/lib/postgresql/data
        # copia o script sql para criacao das tabelas
        - ./bancodedados/create_tables.sql:/docker-entrypoint-initdb.d/create_tables.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
    ports:
        - 25432:5432
    networks:
        - projeto-integrado-network

  frontend:
    container_name: frontend
    build:
      context: ../descomplica
      dockerfile: ./dockerfiles/Dockerfile.React
    ports:
      - "13000:3000"
    networks:
        - projeto-integrado-network
    volumes:
      - ./frontend/node_modules

networks:
    projeto-integrado-network:
        driver: bridge
```

Sobre o arquivo acima, seguem algumas considerações:



- networks: repare que foi criada uma rede específica, "projeto-integrado-network", que é usada em todos os serviços. Isso facilita a comunicação entre o banco de dados x back x front end;

- mapeamento de portas: utilizei portas diferentes das habituais ao mapear, no lado host, cada serviço. Fiz isso para evitar conflitos com outros serviços que, por ventura, rodem no host;

- variáveis de ambiente: no serviço de backend foram setadas as variáveis/propriedades de conexão do spring com o banco de dados, assim como a do JPA/Hibernate (que normalmente são armazenadas no arquivo de propriedades);

- Na chave "volumes", no serviço de banco de dados, foi mapeado o script "create_tables.sql" para o "docker-entrypoint-initdb.d". Essa estratégia permite que o script em questão seja executado na inicialização do container do serviço em questão;

- A entrada Healthcheck é utilizada para checar a saúde do serviço. Com sua utilização, o serviço depende dela (em nosso caso, o backend) aguardar enquanto a saúde do outro serviço (aqui o banco de dados) é checada com sucesso.

Pronto, agora é só subir todo o ambiente. Para isso, execute os comandos a seguir:

a) docker-compose build –no-cache

b) docker-compose up -d

O primeiro comando compila o docker-compose.yml . Caso ocorra algum erro, o mesmo será mostrado no console. Caso executado com sucesso, o segundo comando poderá ser executado. Esse segundo sobe, de fato, todo o ambiente, liberando o terminal (parâmetro "-d"). Caso prefira permanecer visualizando os logs do ambiente, esse comando poderá ser executado sem o último parâmetro.

Nesse ponto, em termos de aplicação, temos alguns fatores limitadores para testarmos cada ambiente. O banco de dados, por exemplo, poderia ser testado com alguma ferramenta como o DBEaver, onde uma conexão para o container deverá ser configurada e então rodarmos algumas queries de sistema, além de conferirmos se as tabelas (do script create_table.sql) foram criadas. Outra forma de analisar se tal serviço subiu com sucesso é analisar os logs do próprio Docker (executando o up sem o "-d"). Sobre o front, além dos logs, é possível abrir o componente default através do navegador, lembrando que deverá ser utilizado o IP da máquina HOST e a respectiva porta (configurada no Docker-compose.yml). Por fim, para testar o backend temos os logs, uma vez que não criamos nenhuma controller, por enquanto. Entretanto, há uma ferramenta disponível que pode nos ajudar nesse sentido e que também é de grande serventia para outras situações. Trata-se do Spring Boot Actuators. Tal biblioteca fornece recursos como monitoramento da saúde da API, fornecimento de métricas, entre outros. Para instalá-la, você deverá incluir a seguinte dependências no pom.xml:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>



</dependency>

Lembre-se: após incluir a dependência, você precisará repetir alguns passos do processo visto até aqui:

- primeiro, precisará parar os contêineres: docker-compose down ;
- depois, precisará fazer o build da aplicação (caso tenha optado por compilar o .jar fora do dockerfile) ;
- por último, deverá subir novamente os contêineres: docker-compose up

Para visualizar a página inicial do Actuator, acesse o endereço http://IP-DO-HOST:PORTA/actuator **(Acesso em 20/10/2022)**. No template do backend que mostrei anteriormente, e que estou usando aqui, foi incluída a dependência de segurança no pom.xml. Logo, se você manteve tal dependência, ao acessar a URL do Actuator verá uma página contendo os campos de usuário e senha. Trata-se do mecanismo default do Spring. Como usuário insira “user” – sem as aspas. Em relação a senha, a mesma é exibida no console pelo Spring, enquanto sobre a aplicação. Nesse caso, para ver tal log, você precisará subir o docker sem o parâmetro “-d”. Copie a senha a partir do console (tomando cuidado para não usar a combinação de teclas CTRL + C, que além de atalho para o comando “copiar”, também é atalho para o comando que libera o terminal, o que faria com que o Docker fosse derrubado, em nosso caso).

Finalmente, ao acessar a URL, você deverá ver uma string JSON contendo alguns dados, como demonstrado na imagem abaix

{"_links":{"self":
{"href":"http://192.168.220.128:18080/actuator","templated":false},"health":
{"href":"http://192.168.220.128:18080/actuator/health","templated":false},"health-path":
{"href":"http://192.168.220.128:18080/actuator/health/{*path}","templated":true}}

Figura 3: Conteúdo da página do Actuator

Repare no conteúdo da página/imagem acima que há algumas URLs listadas. Tratam-se dos endpoints disponíveis. Teste, por exemplo, esse outro endereço: http://IP-DO-HOST:PORTA/actuator/health . **(Acesso em 20/10/2022)**

Após demonstrar como testar a configuração do ambiente dockerizado no qual nosso projeto integrado rodará, chegamos ao final de mais um módulo. Além de tudo o que vimos aqui, você ainda poderia incrementar o processo adicionando pipelines do GitLab, além, é claro, de versionar seu código.

O código fonte da minha aplicação, que serviu como base para a confecção deste módulo, pode ser encontrado aqui:

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo13/aplicacao **(Acesso em 20/10/2022)**

Atividade Extra



Além dos processos aqui demonstrados, há outras formas de dockerizar uma aplicação contendo as mesmas tecnologias usadas aqui. Recomendo que você leia os seguintes links: https://www.baeldung.com/dockerizing-spring-boot-application e https://levelup.gitconnected.com/creating-and-filling-a-postgres-db-with-docker-compose-e1607f6f882f **(Acesso em 20/10/2022)**

Referência Bibliográfica

BAELDUNG. **Running Spring Boot with PostgreSQL in Docker Compose.** Disponível em: <https://www.baeldung.com/spring-boot-postgresql-docker >. **(Acesso em 10/10/2022)**

Atividade Prática Módulo 13

Título da Prática: Alternando o fluxo de dockerização.

Objetivos: Implementar o ambiente do projeto integrado utilizando um fluxo de dockerização alternativo.

Materiais, Métodos e Ferramentas: IDE Spring Tools Suite, Docker.

Atividade Prática

Ao longo do módulo 13 foi apresentado um fluxo de dockerização de todo o ambiente que comporá nosso projeto integrado, composto

pelo banco de dados, backend e frontend. Em relação ao backend foram apresentadas duas formas de realizar a conteinerização do mesmo: compilando previamente o .jar e o copiando, via dockerfile, para o container; copiando todo o código-fonte para o container e realizando o build via dockerfile. Eu utilizei ao longo do módulo o primeiro processo. Nessa atividade prática você deverá utilizar o método diferente do que fez ao longo do módulo. Ou seja, se lá você utilizou o primeiro fluxo, aqui deverá utilizar o segundo e vice-versa.

Lembre-se: não há fluxo certo ou errado, apenas diferentes formas de executar o mesmo processo.

Gabarito Atividade Prática

A execução dessa atividade prática poderá ser feita de duas formas. A seguir são listados os arquivos "Dockerfile" representando ambas as possibilidades de resolução. Através deles o aluno poderá conferir se realizou a atividade de forma correta. Além disso, os passos indicados ao final do módulo também o ajudarão a conferir se todo o processo foi finalizado com sucesso, pois o aluno, por sua conta, poderá aplicar pequenas alterações em relação ao gabarito e ainda assim concluir com sucesso a tarefa.

Seguem os arquivos de gabarito:

https://github.com/FaculdadeDescomplica/pratica_integradora_tecnologias_disruptivas/tree/main/modulo13/atividade-pratica-gabarito

(Acesso em 10/10/2022)



Onde:

- 1.Dockerfile.Spring.Boot é a solução onde o backend é compilado fora do processo de dockerização;
- 2.Dockerfile.Spring.Boot é a solução onde o backend é compilando durante o processo de dockerização;
- Docker-compose.yml: deverá ser o mesmo, independente dos dockerfiles (acima) utilizados.

Ir para exercício